链滴

Ocelot 中文文档一

作者：loogn
原文链接：https://ld246.com/article/1562114204391
来源网站：链滴
许可协议：署名-相同方式共享 4.0 国际 (CC BY-SA 4.0)

入门

Ocelot只能用于.NET Core，目前是为netcoreapp2.0构建的，这个文档可能会帮你了解Ocelot是否合你。

.NET Core 2.0

安装NuGet包

使用nuget安装Ocelot和它的依赖。你需要创建一个netcoreapp2.0的项目并安装Ocelot包。然后按下面的配置部分启动并运行。

```
Install-Package Ocelot
```

所有版本可以在这里找到。

配置

下面是个很简单的ocelot.json。它不做任何事情，但是可以上Ocelot启动了。

```
{
   "ReRoutes": [],
   "GlobalConfiguration": {
      "BaseUrl": "https://api.mybusiness.com"
   }
}
```

这里最重要的就是BaseUrl了。Ocelot需要知道它正在运行的URL，以便头部查找和替换以及一些管配置。设置此URL时，它应该是客户端将看到的Ocelot运行的外部URL。例如，假如你在容器中运行Ocelot的url是http://123.12.1.1:6543，但是在它前面有像nginx一样的代理在响应https://api.mybusinss.com。在这种情况下Ocelot的BaseUrl应该是https://api.mybusiness.com。

如果由于某种原因你使用的是容器，并且希望Ocelot在http://123.12.1.1:6543上响应客户端，那么可以这样做，但如果你正在部署多个Ocelot，你可能会希望通过某种脚本在命令行上传递它。 希望使用的任何调度程序都可以传递这个IP。

程序

然后在你的Program.cs中，你会想要以下内容。主要就是AddOcelot()（添加ocelot服务），UseOcelot().Wait()（设置所有的Ocelot中间件）。

```
public class Program
{
   public static void Main(string[] args)
   {
       new WebHostBuilder()
         .UseKestrel()
         .UseContentRoot(Directory.GetCurrentDirectory())
         .ConfigureAppConfiguration((hostingContext, config) =>
         {
            config
               .SetBasePath(hostingContext.HostingEnvironment.ContentRootPath)
               .AddJsonFile("appsettings.json", true, true)
```

```
                .AddJsonFile($"appsettings.{hostingContext.HostingEnvironment.EnvironmentN
me}.json", true, true)
                .AddJsonFile("ocelot.json")
                .AddEnvironmentVariables();
            })
            .ConfigureServices(s => {
                s.AddOcelot();
            })
            .ConfigureLogging((hostingContext, logging) =>
            {
                //add your logging
            })
            .UseIISIntegration()
            .Configure(app =>
            {
                app.UseOcelot().Wait();
            })
            .Build()
            .Run();
    }
}
```

.NET Core 1.0

原文

不支持

Ocelot不支持一下几点...

- 分块编码 - Ocelot将始终获取body大小并返回Content-Length头。 如果这不适合你的场景，只表示抱歉！
- 转发host头 - 您发给Ocelot的host头不会转发给下游服务。显然这会打破一切 :(
- Swagger - 我已经多次看过从Ocelot的ocelot.json构建swagger.json，但它看起来不适合

我有Ocelot就够了。如果您想在Ocelot中使用Swagger，那么您必须生成自己的swagger.json，并在tartup.cs或Program.cs中执行以下操作。 下面的代码示例注册了一个加载您手动生成的swagger.jso并将其返回到/swagger/v1/swagger.json的中间件。 然后使用Swashbuckle.AspNetCore注册SwagerUI中间件。

```
app.Map("/swagger/v1/swagger.json", b =>
{
    b.Run(async x => {
        var json = File.ReadAllText("swagger.json");
        await x.Response.WriteAsync(json);
    });
});
app.UseSwaggerUI(c =>
{
    c.SwaggerEndpoint("/swagger/v1/swagger.json", "Ocelot");
});
```

```
app.UseOcelot().Wait();
```

我认为Swagger没意义的主要原因是我们已经在ocelot.json中手动编写了我们的定义。如果我们希望对Ocelot开发的人员能够查看可用的路由，那么可以与他们共享ocelot.json（这应该与访问仓库一样单）或者使用Ocelot管理API，以便他们可以查询Ocelot的配置。

除此之外，许多人还会配置Ocelot将所有流量例如/products/{everything}代理到产品服务，如果您析并将其转换为Swagger路径，则不会描述实际可用的内容。另外Ocelot没有有关下游服务可以返回型的概念，并且连接到上述一个端口可以返回多个模型的问题。Ocelot不知道哪些模块需要使用POS还是PUT，所以会变得混乱，最后当swagger.json在运行时发生变化是，Swashbuckle 包不会重新载它。Ocelot的配置可以在运行时更改，这样导致Swagger和Ocelot的信息不匹配。除非我推出我己的Swagger实现。

如果有人想要对Ocelot API 进行简单的测试，我建议使用Postman。甚至可以写一些将ocelot.json射到postman json规范的东西。但是我不打算这样做。

配置

这里有一个配置的列子。其中有两个配置块。一个ReRoutes数组和一个GlobalConfiguration。ReRotes配置块是一些告诉Ocelot如何处理上游请求的对象。Globalconfiguration有些奇特，可以覆盖Reoute节点的特殊设置。如果你不想管理大量的ReRoute特定的设置的话，这将很有用。

```
{
   "ReRoutes": [],
   "GlobalConfiguration": {}
}
```

这是一个ReRoute配置的例子，你不需要全部都设置，但这是目前可用的所有设置：

```
{
        "DownstreamPathTemplate": "/",
        "UpstreamPathTemplate": "/",
        "UpstreamHttpMethod": [
           "Get"
        ],
        "AddHeadersToRequest": {},
        "AddClaimsToRequest": {},
        "RouteClaimsRequirement": {},
        "AddQueriesToRequest": {},
        "RequestIdKey": "",
        "FileCacheOptions": {
           "TtlSeconds": 0,
           "Region": ""
        },
        "ReRouteIsCaseSensitive": false,
        "ServiceName": "",
        "DownstreamScheme": "http",
        "DownstreamHostAndPorts": [
           {
              "Host": "localhost",
              "Port": 51876,
           }
        ],
        "QoSOptions": {
```

```
            "ExceptionsAllowedBeforeBreaking": 0,
            "DurationOfBreak": 0,
            "TimeoutValue": 0
        },
        "LoadBalancer": "",
        "RateLimitOptions": {
            "ClientWhitelist": [],
            "EnableRateLimiting": false,
            "Period": "",
            "PeriodTimespan": 0,
            "Limit": 0
        },
        "AuthenticationOptions": {
            "AuthenticationProviderKey": "",
            "AllowedScopes": []
        },
        "HttpHandlerOptions": {
            "AllowAutoRedirect": true,
            "UseCookieContainer": true,
            "UseTracing": true
        },
        "UseServiceDiscovery": false,
        "DangerousAcceptAnyServerCertificateValidator": false
    }
```

有关如何使用这些选项的更多信息如下..

多环境

和其他任何asp.net core项目一样，Ocelot支持像configuration.dev.json、configuration.test.json的配置文件名。为了实现多环境配置，需要添加如下的代码。

```
.ConfigureAppConfiguration((hostingContext, config) =>
        {
            config
                .SetBasePath(hostingContext.HostingEnvironment.ContentRootPath)
                .AddJsonFile("appsettings.json", true, true)
                .AddJsonFile($"appsettings.{hostingContext.HostingEnvironment.EnvironmentName
.json", true, true)
                .AddJsonFile("ocelot.json")
                .AddJsonFile($"configuration.{hostingContext.HostingEnvironment.EnvironmentNa
e}.json")
                .AddEnvironmentVariables();
        })
```

Ocelot现在将使用特定于环境的配置，如果没有，则返回到ocelot.json。

您还需要设置相应的ASPNETCORE_ENVIRONMENT环境变量。更多信息可以查看asp.net core 文档.

合并配置文件

此功能在问题Issue 296提出要求，并允许用户有多个配置文件，以便管理大型配置。

不直接使用.AddJsonFile("ocelot.json")这样添加配置，还可以使用AddOcelot()方法添加

```
.ConfigureAppConfiguration((hostingContext, config) =>
    {
        config
            .SetBasePath(hostingContext.HostingEnvironment.ContentRootPath)
            .AddJsonFile("appsettings.json", true, true)
            .AddJsonFile($"appsettings.{hostingContext.HostingEnvironment.EnvironmentName}.j
on", true, true)
            .AddOcelot()
            .AddEnvironmentVariables();
    })
```

在这种情况下，Ocelot将查找与模式(?i)ocelot.([a-zA-Z0-9]*).json匹配的任何文件，然后将它们合
。 如果你想设置GlobalConfiguration属性，你必须有一个名为ocelot.global.json的文件。

Ocelot合并文件的方式基本上是加载、循环它们，添加所有ReRoutes，添加所有AggregateReRoute，如果文件名为ocelot.global.json，则添加GlobalConfiguration以及所有ReRoutes或AggregateRRoutes。 然后Ocelot将合并后的配置保存到一个名为ocelot.json的文件中，这才是ocelot运行时配的真正来源。

目前在这个阶段没有验证，只有当Ocelot最终合并配置时才会发生验证。 当你排查问题时要注意这点。如果您有任何问题，建议您始终检查ocelot.json中的内容。

将配置存储在consul中

如果您在注册服务时添加以下内容，Ocelot将尝试在consul的KV数据中存储和检索其配置。

```
services
   .AddOcelot()
   .AddStoreOcelotConfigurationInConsul();
```

您还需要将以下内容添加到您的ocelot.json中。 这使Ocelot知道如何找到您的Consul代理并进行交
以及从Consul加载和存储配置。

```
"GlobalConfiguration": {
    "ServiceDiscoveryProvider": {
        "Host": "localhost",
        "Port": 9500
    }
}
```

我在研究过raft算法并发现其超级之难后决定创建此功能。 为什么不利用Consul已经给你的这个便利！ 我想这意味着如果你想最大限度地使用Ocelot，你现在就需要将Consul作为依赖。

此功能在向本地consul代理发出新请求之前有3秒TTL缓存。

重定向 / 使用CookieContainer

在ReRoute配置中使用HttpHandlerOptions来设置HttpHandler行为：

1. AllowAutoRedirect的值指示请求是否应该遵循重定向响应。 如果请求应该自动重定向来自下游资的响应，请将其设置为true; 否则为false。 默认值是false。

2. UseCookieContainerd的值指示处理程序是否使用CookieContainer属性存储服务器cookie，并发送请求时使用这些cookie。默认值是false。值得注意的是，如果您使用CookieContainer，Ocelo为每个下游服务缓存HttpClient。这意味着对该下游的所有请求都将共享相同的Cookie。 问题274的出是因为用户观察到Cookie被共享了。我试图想出一个好的解决方案，但我认为是不可能的。如果您缓存客户端，这意味着每个请求都获得一个新客户端并因此获得一个新的cookie容器。如果您清除缓客户端容器中的Cookie，则会因为正在进行的请求而竞争。这也意味着后续请求不能使用以前响应中cookie！总而言之，这不是一个好的局面。我会避免将UseCookieContainer设置为true，除非有一非常好的理由。只需查看您的响应头，并在您的下一个请求把cookie转发回来！

SSL错误

如果您想忽略SSL警告/错误，请在ReRoute配置中设置以下内容。

```
"DangerousAcceptAnyServerCertificateValidator": false
```

我不建议这么做，如果可以我建议你创建自己的证书，然后让本地/远程的机器信任它。

路由

Ocelot的主要功能是接管进入的http请求并把它们转发给下游服务。目前是以另一个http请求的形式将来可能是任何传输机制）。

Ocelot将路由一个请求到另一个请求描述为ReRoute。为了在Ocelot做任何工作，都需要在配置中置一个ReRoute。

```
{
   "ReRoutes": [
   ]
}
```

为了设置ReRoute，你需要如下所示添加一个ReRoute到ReRoutes的json数组。

```
{
   "DownstreamPathTemplate": "/api/posts/{postId}",
   "DownstreamScheme": "https",
   "DownstreamHostAndPorts": [
         {
            "Host": "localhost",
            "Port": 80,
         }
      ],
   "UpstreamPathTemplate": "/posts/{postId}",
   "UpstreamHttpMethod": [ "Put", "Delete" ]
}
```

DownstreamPathTemplate、DownstreamScheme和DownstreamHostAndPorts 确定请求的转。

DownstreamHostAndPorts是一个数组，包含请求要转发的主机和端口。通常这只包含一个条目，有时您可能需要将请求负载平衡到您的下游服务，这是Ocelot允许我们添加多个条目，然后选择一个载均衡器。

UpstreamPathTemplate是Ocelot用来标识哪个DownstreamPathTemplate用于给定的请求URL。

后，UpstreamHttpMethod的使用，可以让Ocelot区分对同一个URL的请求，并且显然这是需要的作。

你可以指定一个Http请求方法列表，或者一个空的列表以允许任务Http请求方法。在Ocelot中，你以使用{something}的方式在模板中添加变量占位符。占位符需要在DownstreamPathTemplate 和UstreamPathTemplate中都添加。如果是这样，当请求到达时Ocelot将试图使用上游url中的正确的变值来替换占位符。

你也可以想这样使用一个ReRoute处理所有请求：

```
{
   "DownstreamPathTemplate": "/api/{everything}",
   "DownstreamScheme": "https",
   "DownstreamHostAndPorts": [
         {
            "Host": "localhost",
            "Port": 80,
         }
      ],
   "UpstreamPathTemplate": "/{everything}",
   "UpstreamHttpMethod": [ "Get", "Post" ]
}
```

这将转发所有请求到下游服务路径/api。

目前在没有任何配置的请求下，Ocelot将默认所有ReRoutes不区分大小写。 为了改变这种情况，您以在每个ReRoute中指定以下设置：

```
"ReRouteIsCaseSensitive": true
```

这意味着，当Ocelot尝试将上行url与上游模板匹配时将区分大小写。 此设置默认为false，这也是我建议。因此只有在您希望ReRoute区分大小写时才用设置它。

捕获所有

Ocelot的路由还支持捕获所有样式的路由，用户可以指定他们想要匹配所有流量。如果你像下面那样置你的配置，请求将被直接代理（它不一定叫url，任何占位符名称都可以）。

```
{
   "DownstreamPathTemplate": "/{url}",
   "DownstreamScheme": "https",
   "DownstreamHostAndPorts": [
         {
            "Host": "localhost",
            "Port": 80,
         }
      ],
   "UpstreamPathTemplate": "/{url}",
   "UpstreamHttpMethod": [ "Get" ]
}
```

该捕获所有的优先级低于其他任何ReRoute。 如果你的配置中还有下面的ReRoute，那么Ocelot会捕获所有配置之前先匹配它。

```
{
   "DownstreamPathTemplate": "/",
   "DownstreamScheme": "https",
   "DownstreamHostAndPorts": [
         {
            "Host": "10.0.10.1",
            "Port": 80,
         }
      ],
   "UpstreamPathTemplate": "/",
   "UpstreamHttpMethod": [ "Get" ]
}
```

上游主机

此功能允许您基于上游主机进行ReRoutes。 这是通过查看客户端使用的主机头来工作，然后将其用识别ReRoute的信息的一部分。

为了使用这个功能，在你的配置中加上如下配置。

```
{
   "DownstreamPathTemplate": "/",
   "DownstreamScheme": "https",
   "DownstreamHostAndPorts": [
         {
            "Host": "10.0.10.1",
            "Port": 80,
         }
      ],
   "UpstreamPathTemplate": "/",
   "UpstreamHttpMethod": [ "Get" ],
   "UpstreamHost": "somedomain.com"
}
```

上面的ReRoute只会匹配主机头是somedomain.com的请求。

如果您没有在ReRoue上设置UpstreamHost，则任何主机头都可以匹配它。 这基本上是一个捕获所功能并保留构建功能时的现有功能。这意味着如果您有两个相同的ReRoute，其中一个与UpstreamHst是null，另一个有值。 Ocelot会倾向于设定值的那个。

这个功能在问题 216提出要求。

优先级

在问题 270中，我最终决定在ocelot.json中公开ReRoute的优先级。这意味着您可以决定上游HttpReuest与你的ReRoutes的匹配顺序。

为了是其起作用，将以下内容添加到ocelot.json的ReRoute中，0仅仅是一个示例值，将在下面解释。

```
{
   "Priority": 0
}
```

0是最低优先级，Ocelot将始终使用0作为/{catchAll}路由条目，并且可以硬编码。之后，你可以自由置你想要的任何优先级。

例如你可以这样：

```
{
   "UpstreamPathTemplate": "/goods/{catchAll}"
   "Priority": 0
}
```

还可以：

```
{
   "UpstreamPathTemplate": "/goods/delete"
   "Priority": 1
}
```

在上面的例子中，如果您向Ocelot请求/goods/delete，Ocelot将匹配/goods/delete这个ReRoute不过在不设置优先级以前它会匹配/goods/{catchAll}（因为这是列表中的第一个ReRoute！）。

请求聚合

Ocelot允许您指定聚合多个普通ReRoutes的Aggregate ReRoutes（聚合路由），并将其响应映射一个对象中。一般用于当您有一个客户端向服务器发出多个请求，而这些请求可以合并成一个的时候此功能允许您通过Ocelot实现前端类型结构的后端。

此功能是问题 79的一部分，并且作为问题 298的一部分进行了进一步改进。

为了设置它，你必须在ocelot.json中做如下的事情。 这里我们已经指定了两个普通的ReRoutes，每个都有一个Key属性。 然后，我们使用ReRouteKeys列表中的键指定组成两个ReRoutes的聚合，然设置UpstreamPathTemplate，它的工作方式与普通的ReRoute相似。 很明显，您不能在ReRoutes Aggregates之间复制UpstreamPathTemplates。 除RequestIdKey之外，您可以使用普通ReRoute有的选项（在下面的陷阱中进行了解释）。

高级应用-注册你自己的聚合器

Ocelot只是基本的请求聚合，然后我们添加了一个更高级的方法，让用户从下游服务中获取响应，然将它们聚合到响应对象中。

ocelot.json的设置与基本聚合方法几乎相同，只需额外添加一个Aggregator属性，如下所示。

```
{
   "ReRoutes": [
      {
         "DownstreamPathTemplate": "/",
         "UpstreamPathTemplate": "/laura",
         "UpstreamHttpMethod": [
            "Get"
         ],
         "DownstreamScheme": "http",
         "DownstreamHostAndPorts": [
            {
               "Host": "localhost",
```

```
                "Port": 51881
            }
        ],
        "Key": "Laura"
    },
    {
        "DownstreamPathTemplate": "/",
        "UpstreamPathTemplate": "/tom",
        "UpstreamHttpMethod": [
            "Get"
        ],
        "DownstreamScheme": "http",
        "DownstreamHostAndPorts": [
            {
                "Host": "localhost",
                "Port": 51882
            }
        ],
        "Key": "Tom"
    }
  ],
  "Aggregates": [
    {
        "ReRouteKeys": [
            "Tom",
            "Laura"
        ],
        "UpstreamPathTemplate": "/",
        "Aggregator": "FakeDefinedAggregator"
    }
  ]
}
```

这里我们添加了一个叫FakeDefinedAggregator的聚合器。当Ocelot尝试聚合这个ReRoute的时候会去查看这个聚合器。

为了使这个聚合器可用，我们必须像下面这样把FakeDefinedAggregator添加到OcelotBuilder。

```
services
  .AddOcelot()
  .AddSingletonDefinedAggregator<FakeDefinedAggregator>();
```

现在，当Ocelot尝试聚合上述ReRoute时，它会在容器中找到FakeDefinedAggregator并使用它来合ReRoute。 由于FakeDefinedAggregator是在容器中注册，因此您可以将它需要的任何依赖项都加到容器中，如下所示。

```
services.AddSingleton<FooDependency>();

services
  .AddOcelot()
  .AddSingletonDefinedAggregator<FooAggregator>();
```

在这个例子中FooAggregator依赖FooDependency，将会被容器解析。

除此之外，Ocelot还允许您添加如下所示的瞬态聚合器。（参考.net core依赖注入，译者注）

```
services
    .AddOcelot()
    .AddTransientDefinedAggregator<FakeDefinedAggregator>();
```

为了实现一个聚合器，你必须实现这个接口。

```
public interface IDefinedAggregator
{
    Task<DownstreamResponse> Aggregate(List<DownstreamResponse> responses);
}
```

使用此功能，您几乎可以做任何您想做的事情，因为DownstreamResponse包含内容，头和状态代。 如果需要，我们可以添加额外的东西，只需在GitHub上提出这个问题。请注意，如果在向聚合中的eRoute发出请求时HttpClient抛出异常，那么您将不会获得其DownstreamResponse，但您会获得他请求成功的DownstreamResponse。 如果某个请求抛出异常，则会被记录。

基本演示

```
{
    "ReRoutes": [
        {
            "DownstreamPathTemplate": "/",
            "UpstreamPathTemplate": "/laura",
            "UpstreamHttpMethod": [
                "Get"
            ],
            "DownstreamScheme": "http",
            "DownstreamHostAndPorts": [
                {
                    "Host": "localhost",
                    "Port": 51881
                }
            ],
            "Key": "Laura"
        },
        {
            "DownstreamPathTemplate": "/",
            "UpstreamPathTemplate": "/tom",
            "UpstreamHttpMethod": [
                "Get"
            ],
            "DownstreamScheme": "http",
            "DownstreamHostAndPorts": [
                {
                    "Host": "localhost",
                    "Port": 51882
                }
            ],
            "Key": "Tom"
        }
    ],
    "Aggregates": [
        {
            "ReRouteKeys": [
```

```
            "Tom",
            "Laura"
        ],
        "UpstreamPathTemplate": "/"
    }
  ]
}
```

你也可以设置Aggregate的UpstreamHost和ReRouteIsCaseSensitive，和其他ReRoutes的作用是样的。

如何路由/tom返回 {“Age”：19}，路由/laura返回{“Age”：25}，那么聚合之后的相应就如下所示。

```
{"Tom":{"Age": 19},"Laura":{"Age": 25}}
```

目前的聚合功能非常简单。 Ocelot只是从你的下游服务获得响应，并将其复制到json字典中，如上示。将ReRoute键作为字典的关键字，下游服务的响应体作为值。你可以看到这个对象就是没有任何进空格的JSON。

来自下游服务相应的所有头部都会丢失。

Ocelot将总是将聚合请求的内容类型返回application/json。

如果下游服务返回404，那么聚合将为该下游服务返回空内容。 即使所有下游都返回404，它也不会聚合响应为404。

疑难杂症 / 更多信息

您不能将ReRoutes与特定的RequestIdKeys一起使用，因为这将使跟踪非常的复杂。

聚合仅支持GET HTTP动词。